Hypervisor Development Hands On for Security Researchers on Windows

VXCON 2019 Version

Satoshi Tanda

Disclaimer

- Please do not share the contents (slides, source code etc)

Outline & Time schedule

- Introduction (10:00-10:30)
  - Goals and Non-Goals
  - Review of Prerequisites Specifics
  - Terminology
  - Exercise 0: Run Our Hypervisor and Observe Behavior
- Basics of Intel VT-x (10:30-12:00)
- Lunch break (12:00-13:00)
- Customizing Hypervisor (13:00-14:30)
- Wrapping Up (14:30-15:00)

Who I am

- Software Engineer at CrowdStrike
  - 10 years of professional experience in Windows system programming
- Reverse Engineer
  - Ex-vulnerability and malware analyst at GE & Sophos
- Twitter: @standa_t
- Speaker: Recon, BlueHat, Nullcon, CodeBlue
- Hiker, camper, runner, diver, cat lover, husband etc

- Creator of HyperPlatform & SimpleSvm – open source hypervisors on Windows for Intel and AMD

Goals of workshop

- Familiarize yourself enough with concepts and implementation of hypervisor to start exploring further details by yourselves
  - Understand the key concepts of VT-x
  - Understand how VT-x is controlled from C-code on Windows
- How
  - Knowledge: Explanation of concepts
  - **Exercise: Following code implementing concepts**
  - References: Sharing pointers for further learning

Non-goals / out-of-scope

- Coverage of AMD-V
- X86 (32bit) specific factors
- Support of non-Windows OS
- About Windows kernel-mode driver development and the Windows internals
- Some more details and advanced features
  - Use of Extended Page Tables (EPT)
- Isolation of the hypervisor from the guest
- Booting Windows under hypervisor

Prerequisites

- Windows laptop as instructed here:
  - https://tandasat.github.io/VXCON/
- Fluency in C
- Understanding of the basics of IA32 architecture
- Familiarity with Windows kernel-mode driver development
  - Visual Studio, Windbg

Hypervisor we develop

- Blue Pill style; neither type-1 nor type-2
- Just a driver file
- Virtualize the current OS
  - Hypervisor monitors activities of the host OS
  - Do not create & boot a new instance of OS (VM)

- Think of it as the driver that enables additional interrupts and registers handlers
  - Interrupts = VM-exit
  - Handlers = Hypervisor

Exercise 0: Run Our Hypervisor and Observe Behavior

- Download the precompiled hypervisor
- Follow the instructions to install it
- Password: 12345

That’s what we will code

- Hypervisor written in C
  - Enables VM-exit (interrupts) for CPUID
  - Handles it and debug prints parameters
  - Will be extended in the afternoon exercise
- Small
  - Less than 3000 lines (and 40% of them are comments)
  - Heavily commented with references to the Intel manual
  - 1000 lines of them are just logger code
    - = 1100 lines of hypervisor ☺

Basics of Intel VT-x

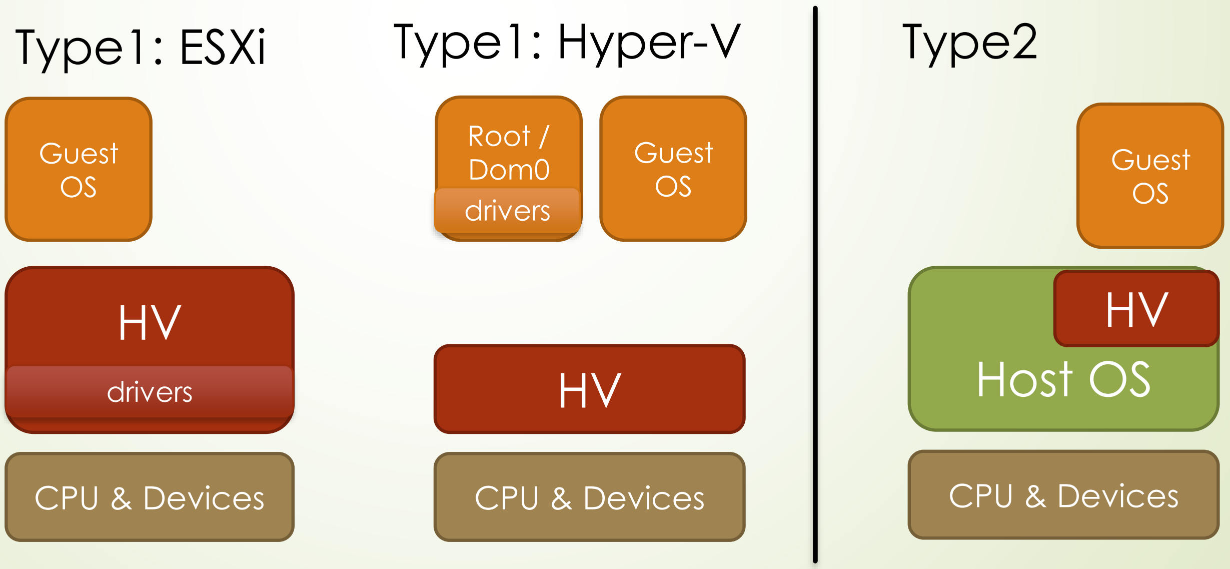
Types of virtualization

- Language-based VM: Java, C#.
- Lightweight VM: Docker, Google Native Client, Jail.
- **System-level VM**
  - Emulator: Does not execute instructions directly, instead, emulates them. Eg, Bochs
  - Software Virtualization: Directly executes instructions but trap-and-emulates some **without** hardware assistance. Eg, old VMware
  - **Hardware Virtualization**: Directly executes instructions but trap-and-emulates some **with** hardware assistance. Eg, KVM
  - Paravirtualization: Directly executes instructions but modifies the guest to isolate it. Eg, old Xen

Hypervisor

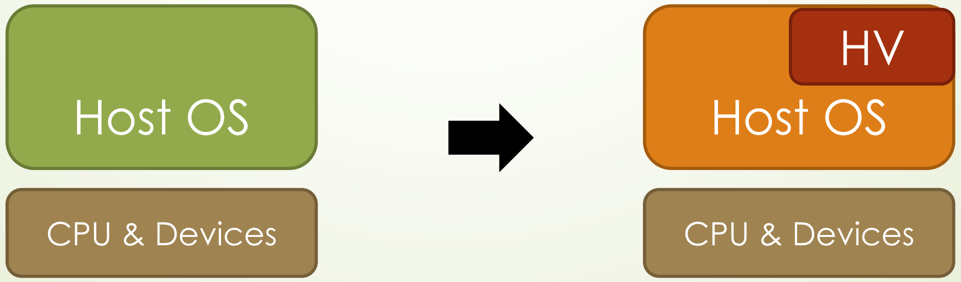
Types of hypervisor

- Type 1: Runs on bare metal. Eg, VMware ESXi, Hyper-V
- Type 2: Runs on the host OS. Eg, VMware Workstation



Our hypervisor

- The virtual machine is carefully created illusion with trap-and-emulation approach for the "guest"
  - With hardware virtualization, this can be applied for the host
- **Blue Pill style**: Runs on the host OS. Virtualizes the host. Eg, BluePill, SimpleVisor, HyperPlatform, SimpleSVM.
  - Does not provide isolated "guest"



Terminology 1/2

- Hypervisor (HV): System software that provides virtualized environment and manages virtual machines in it.
- Virtual Machine Monitor (VMM): Often equivalent to hypervisor. Intel SDM defines and uses this term.
  - See: 23.2 VIRTUAL MACHINE ARCHITECTURE.
- Host OS: OS that is running with a type 2 hypervisor. This is a confusing term from the technical standpoint, and largely irrelevant to us. Therefore, not used in this class.
- Guest OS: OS that is running on the top of virtual HW. This is mostly an irrelevant concept for us; therefore, not used in this class.

Terminology 2/2

- Host: equivalent to hypervisor. Often used to contrast with "guest". Intel SDM refers to hypervisor as host.
  - In our class, think of code in VmExit.c. The rest is “set up code”, ie, not all code is host.
- Guest: execution context that is governed by the hypervisor.
  - On full fledged hypervisors, this is a virtual machine made up of a set of virtual devices, OS and applications running on the top of it.
  - In our cases, this is everything but hypervisor.

Entering VMX operation

- What does "enable VM-exit" really mean?
- Hardware virtualization technology on the Intel processor is called VT-x.
  - VT-x needs to be enabled by system platform software
  - VT-x needs to be activated by software
  - When VT-x is activated, it is called that processor is in **VMX operation**, which enables some more instructions
- Our diver executes the **VMXON** instruction
  - This itself does not enable VM-exit yet

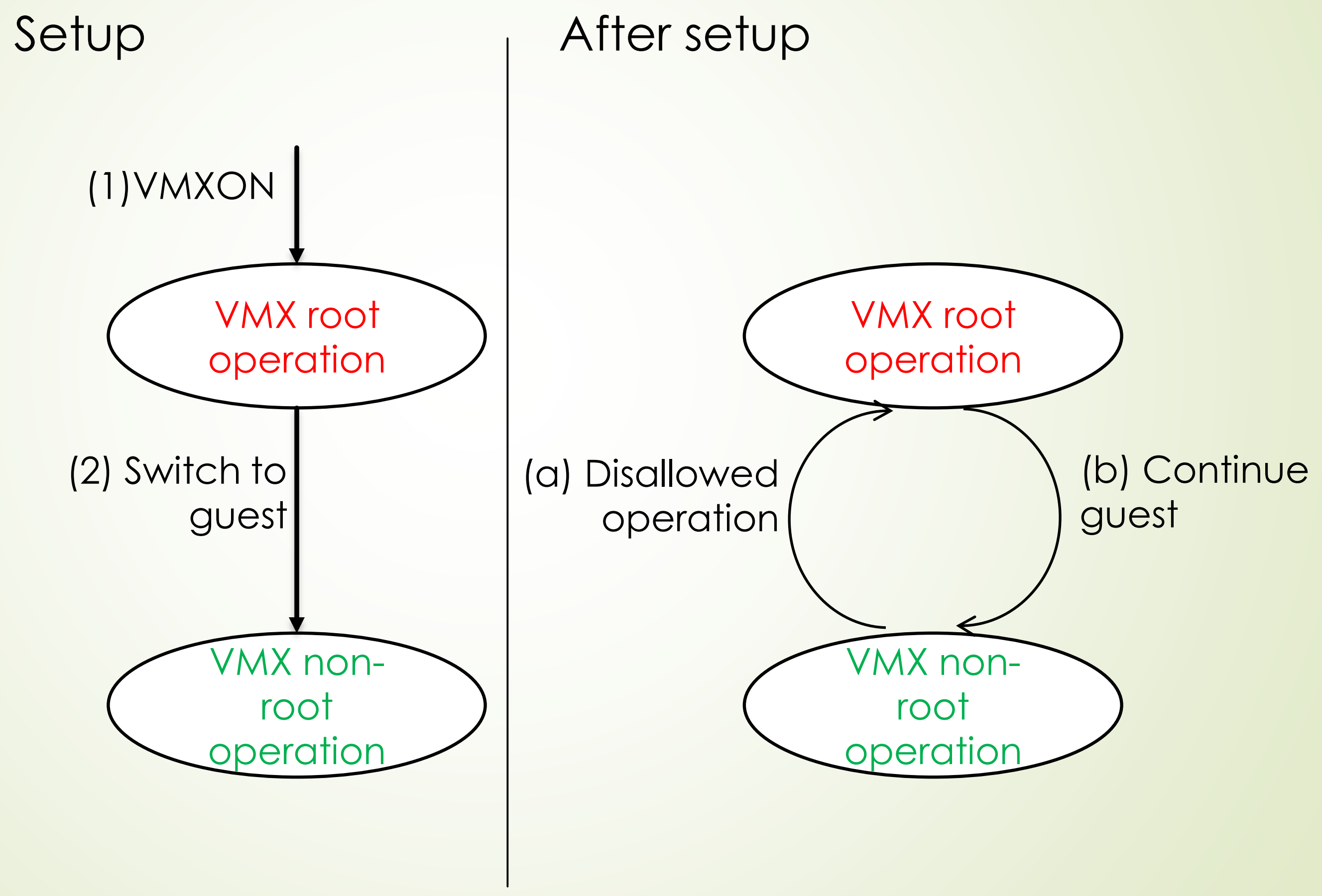
Root vs non-root operation

- **VMX root operation**: privileged and responsible for managing VMs. The initial state after VMXON.
- **VMX non-root operation**: unprivileged in that some operations trigger “interrupts”. This is where non-hypervisor code runs.
  - This “interrupt” transitions mode to VMX root operation to handle the operation
  - The hypervisor is responsible for handling the operation and runs in VMX root operation
  - Once the hypervisor handles the operation, it transitions mode to VMX non-root operation to let the original context to continue to run

Host-guest interaction

- Flow
  1. We start as a root, set up both guest and host, switch to the guest
  2. The guest runs as non-root, until it attempts to perform a trapped operation
  3. Our host (hypervisor) runs as root, emulates the operation, and switch back to the guest
- The **host can do more than just pure emulation**
  - Eg, logging, altering results, or even changing guest's RIP

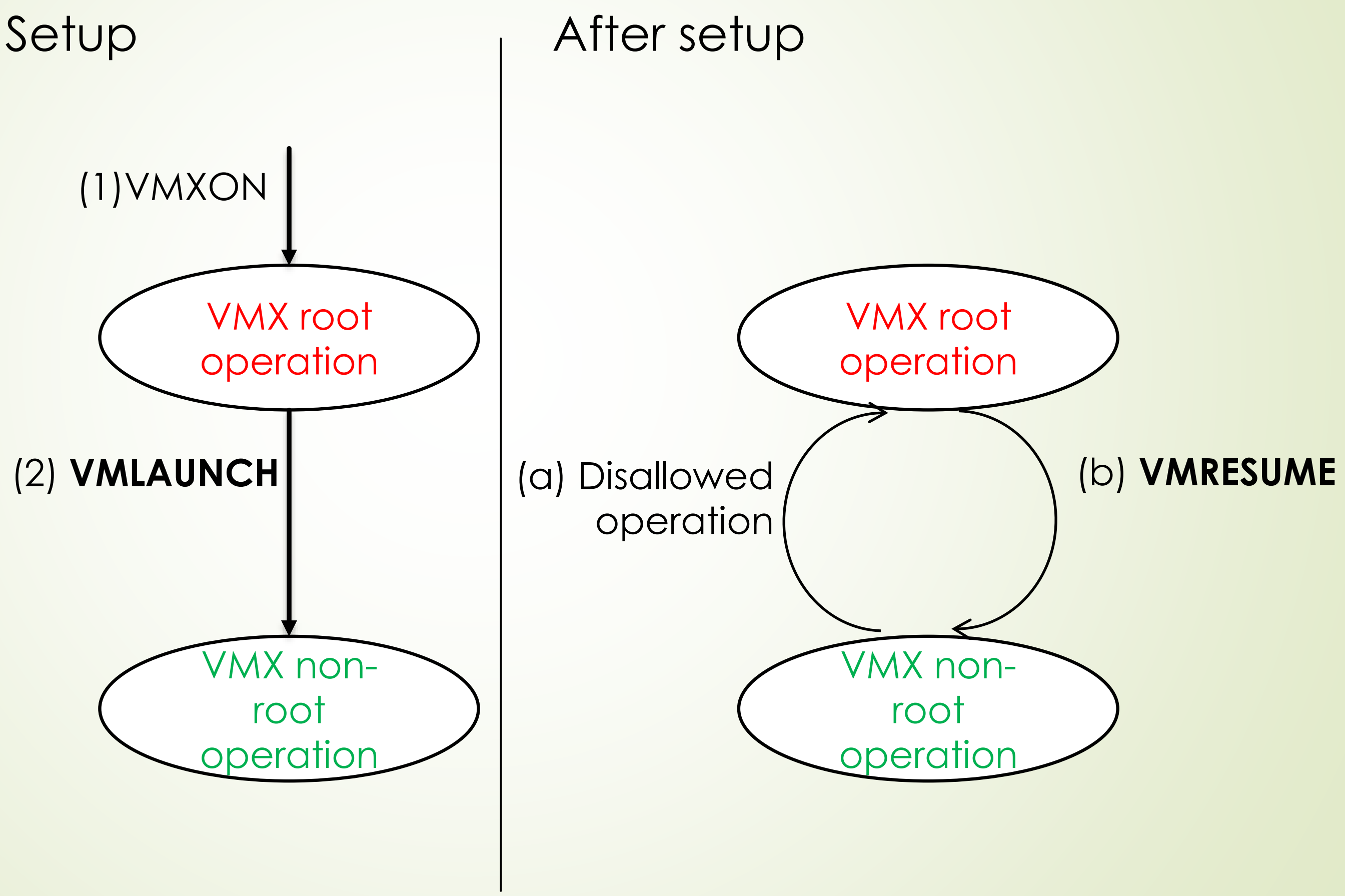
Entering VMX and Flow



Transition and Instructions

- Initial state after VMXON is VMX root operation
  - This enables few more VMX instructions to set up both guest and host (details later)
- After setting up host and guest, our driver executes the **VMLAUNCH** instruction
  - This starts running the guest as set up, under VMX non-root operation
- The host handles some of guest's operations. After handling them, the host executes the **VMRESUME** instruction to go back to the guest

Entering VMX and Flow 2



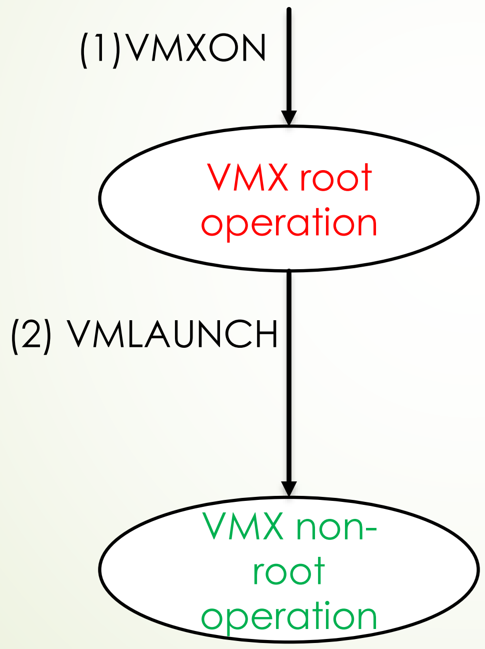
VM-exit and VM-enter

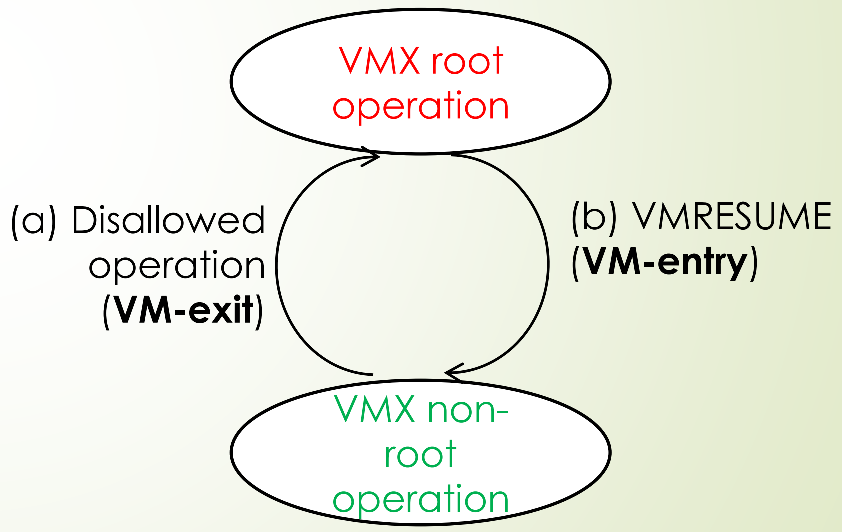
- **VM-exit**:
  - VMX non-root operation => VMX root operation
  - What we referred to as "interrupt" is VM-exit
  - Triggered by various operations in the guest
- **VM-entry**:
  - VMX root operation => VMX non-root operation
  - VMRESUME from the host perform this

Entering VMX and Flow 3

Setup

After setup




VMX root operation
(a) Disallowed operation (**VM-exit**)
(b) VMRESUME (**VM-entry**)
VMX non-root operation

VMCS: The Context Structure for VT-x

Setting up host and guest

- What does this mean exactly?
  1. Allocate 4KB of structure called VMCS
  2. Instruct the processor to use the VMCS for VMX instructions and VM-exit/-entry
  3. Filling out the VMCS to set up:
     - Initial guest state (eg, register values)
     - Initial host state
     - When VM-exit to occur (eg, should MOV-to-CR3 trigger VM-exit?)
     - What other VT-x features to enable (eg, EPT)

Accessing VMCS

- Used to configure behavior of VT-x, as well as buffer to save and load host and guest states
- Opaque structure, must reside in physical memory
  - Access must be done with special instructions, except for the first 32bit (ie, structure version)
  - Eg, Initialization is done with the **VMCLEAR** instruction

```
unsigned char __vmx_vmclear(
  unsigned __int64 *VmcsPhysicalAddress
);
```

Read from and Write to VMCS

- Read and write must be done with the **VMREAD** and **VMWRITE** instructions
  - Those instructions take a "Field" value to indicate what fields to read from and write to, instead of an offset

```
unsigned char __vmx_vmwrite(
  size_t Field,
  size_t FieldValue
);
```

  - Over 170 fields are defined; we use about 70

Current VMCS

- Notice VMREAD and VMWRITE do not take a parameter to indicate what VMCS to read from and write to
- Instead, those always access the **current VMCS**
- The **VMPTRLD** instruction takes VMCS and set it as the current VMCS
  - Must be executed before use of VMREAD and VMWRITE

```
unsigned char __vmx_vmptrld(
  unsigned __int64 *VmcsPhysicalAddress
);
```

Review

1. Allocate 4KB of structure called VMCS
   - `VmcsRegion = ExAllocatePoolWithTag(NonPagedPool, …);`
2. Instruct the processor to use the VMCS for VMX instructions and VM-exit/-entry
   - `VmcsRegion->RevisionId = … // taken from MSR`
   - `vmcsPa = GetPhysicalAddress(VmcsRegion);`
   - `__vmx_vmclear(&vmcsPa);`
   - `__vmx_vmptrld(&vmcsPa);`
3. Filling out the VMCS to set up the host and guest
   - `__vmx_vmwrite(…);`

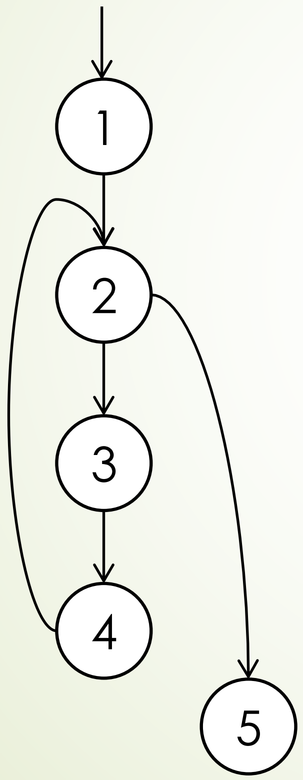
Setting up the guest state

- VMCS has fields to specify the initial state of the guest
- The processor loads those field values on VMLAUNCH and VM-entry
  - Similar to process context switch, where software takes new process's state from memory and put them into the processor
  - Also similar to UM->KM transition, where the processor takes kernel-mode system registry values from MSRs and put them into the processor
- What's guest's initial state for us?

Setting up the guest state

- In our class, the guest (VM) is the current system
- **Guest state in VMCS is pure mirror of the current state**
- Once VMLAUNCH is executed, the processor just keeps running under the current state, except the processor is in VMX non-root operation
  - Think of it as calling setjmp and longjmp immediately after
    - Where setjmp is sequence of VMWRITE, and longjump is VMLAUNCH

Guest state and VMLAUNCH



1. Copy current state
2. Check if the system is in VMX non-root operation
3. if not,
   - enter VMX root operation (VMXON)
   - capture more current state, and
   - set up the guest (VMWRITE)
4. Execute VMLAUNCH, which jumps to right after (1) but in VMX non-root operation
5. Let the system run as if nothing happened

Setting up the host state

- VMCS has fields to specify the state of the host
- The processor loads those field values on VM-exit
- What's guest's initial state for us?

Setting up the host state

- In our class, the host (hypervisor) runs part of the current system
- Host state is in VMCS is mostly **mirror of the current state, except RSP and RIP**
  - RIP is set to the entry point to hypervisor logic
  - RSP is set to scratch space
- When VM-exit occurs, the processor updates its RIP to start running hypervisor code
  - Think of this as interrupts
- Hypervisor can behave like a regular driver as system register values are mirrored, except the processor is in VMX root-operation

Review – Guest Startup

(1) VMCLEAR

(2) VMPTRLD

(3) VMWRITE → GUEST_CR0

VMWRITE → GUEST_CR3

VMWRITE → GUEST_RIP

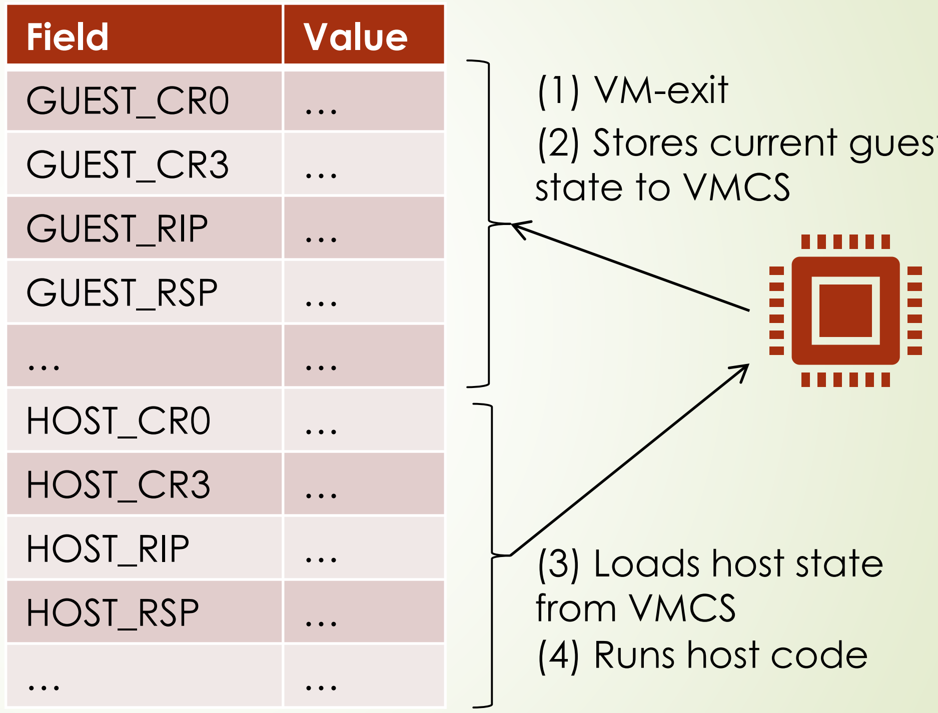
…

Field	Value
GUEST_CR0	…
GUEST_CR3	…
GUEST_RIP	…
GUEST_RSP	…
…	…
HOST_CR0	…
HOST_CR3	…
HOST_RIP	…
HOST_RSP	…
…	…

(4) VMLAUNCH --
Loads guest state from VMCS

(5) Runs guest code

Review – VM-exit



Field	Value
GUEST_CR0	…
GUEST_CR3	…
GUEST_RIP	…
GUEST_RSP	…
…	…
HOST_CR0	…
HOST_CR3	…
HOST_RIP	…
HOST_RSP	…
…	…

(1) VM-exit
(2) Stores current guest state to VMCS
(3) Loads host state from VMCS
(4) Runs host code

Review – VM-entry

Field	Value
GUEST_CR0	…
GUEST_CR3	…
GUEST_RIP	…
GUEST_RSP	…
…	…
HOST_CR0	…
HOST_CR3	…
HOST_RIP	…
HOST_RSP	…
…	…

(1) VMRESUME (VM-entry)

(2) Loads guest state from VMCS

(3) Runs guest code

Code walkthrough

Minivisor

- Open exercise1 (password: 23456)
- A hypervisor written specifically for education
- Ia32-doc defines most Intel SDM defined structures and constants
  - https://github.com/wbenny/ia32-doc
  - Author: @PetrBenes

Initialization flow

- DriverEntry
  - EnableHypervisorOnAllProcessors
    - EnableHypervisor
      - AsmGetCurrent(Stack | Instruction)Pointer
      - IsOurHypervisorInstalled
      - PrepareForVmLaunch
      - VMLAUNCH

VM-exit flow

- AsmHypervisorEntryPoint
  - HandleVmExit
    - Handle*
  - VMRESUME

Details covered later

- VM-entry and VM-exit control fields in VMCS – all VMCS_CTRL_xxx

Handing VM-exit

- On VM-exit, the processor saves its reason in VMCS
- The host reads it and typically does one of those:
  - Emulate the attempted operation on behalf of the guest
    - Includes updating guest's registers and injecting exceptions
    - Eg, CPUID
  - Update underneath configurations and let the guest retry
    - Eg, EPT-violation
- Then, executes VMRESUME

Gochas

- On VM-exit, the host must store some of guest's state manually
  - The processor does not save values of general purpose and XMM registers into VMCS on VM-exit
  - Changes of those values during host-execution directly reflects to the guest, and corrupts guest state
- Before VMRESUME, those must be loaded manually

Gochas

- On VM-exit, the processor disables interrupt by clearing eflags.IF
  - Otherwise, the processor may run any unintended code in VMX root-operation
- This prevents call of all NT API but ones allowed at HIGH_LEVEL
  - Page-fault cannot be handled
  - Timer interrupt cannot be executed
  - IPI cannot be processed

Exercise 1: Implement the CPUID handler

- The provided code lacks implementation of CPUID emulation
  - Hypervisor will crash if executed due to unhandled VM-exit
- Implement the CPUID handler in VmExit.c
  - Remember that emulated CPUID should return what IsOurHypervisorInstalled expects
  - Study the RDMSR handler as a reference
- Tips:
  - Use the MiniVisor\DumpActiveLogs.js to dump logs
  - Activate the breakpoints that are commented out
  - Do this to disable those breakpoints: kd> eb @rip 90

Solution

- Open the solution1 (password: 23456)
- The host call VMREAD and checks whether the reason is CPUID
- In that case, executed the CPUID instruction as the guested attempted
  - Use guest registers
- Update the guest register values on stack based to reflect the result
  - But uses the modified result in case of CPUID_HV_VENDOR_AND_MAX_FUNCTIONS
- Advances guest's RIP

Customizing Hypervisor

Control fields in VMCS

- VMCS has fields to control behavior of VT-x
  - VM-exit / -entry control fields: control what processor does on VM-exit and VM-entry
    - Eg, Enabling 64bit (x86-64) mode on transition as necessary
    - See tables in 24.7.1 VM-Exit Controls / 24.8.1 VM-Entry Controls
  - VM-execution control fields: mainly control when VM-exit to occur
    - Eg, Setting up which MSR access to cause VM-exit via bitmaps, and whether MOV-to-CR3 causes VM-exit
    - See: 24.6.9 MSR-Bitmap Address / 24.6.2 Processor-Based VM-Execution Controls

VM-exit information fields

- Read-only VMCS fields; the processor stores information about the latest VM-exit, for example,
  - **Exit reason** stores the reason of VM-exit
  - **Exit qualification** stores the additional information. Written only for some VM-exit and formats are reason-dependent
- Eg, how can the host do emulation with only information like "*Guest software attempted to access CR0, CR3, CR4, or CR8 using CLTS, LMSW, or MOV CR*" ?
  - Requires disassembling code to know exactly what instruction caused VM-exit and how to emulate it
- Exit-qualification gives enough information

Example exit qualification

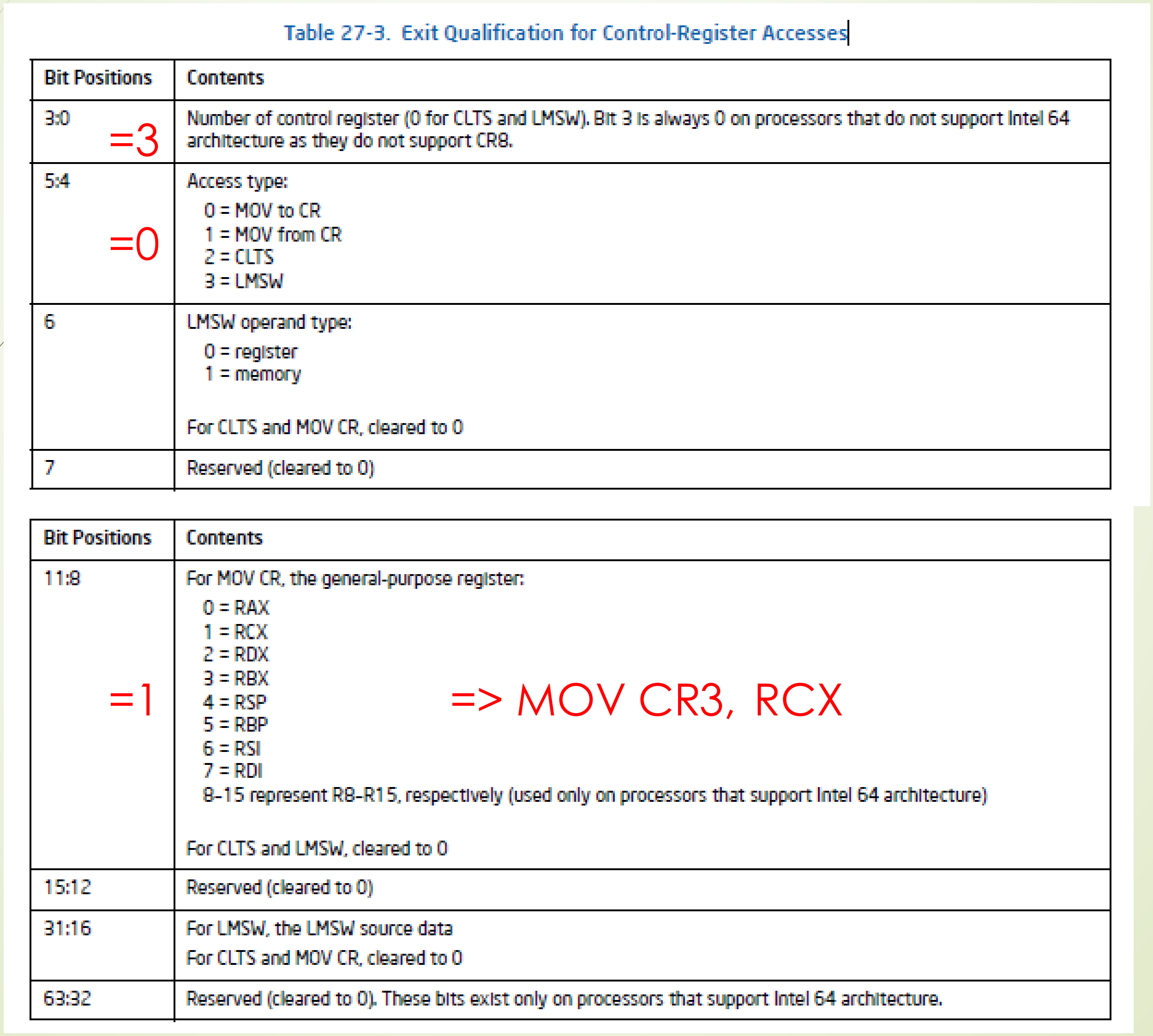


Table 27-3. Exit Qualification for Control-Register Accesses

Bit Positions	Contents
3:0 =3	Number of control register (0 for CLTS and LMSW). Bit 3 is always 0 on processors that do not support Intel 64 architecture as they do not support CR8.
5:4 =0	Access type: 0 = MOV to CR 1 = MOV from CR 2 = CLTS 3 = LMSW
6	LMSW operand type: 0 = register 1 = memory For CLTS and MOV CR, cleared to 0
7	Reserved (cleared to 0)
11:8 =1	For MOV CR, the general-purpose register: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) For CLTS and LMSW, cleared to 0 => MOV CR3, RCX
15:12	Reserved (cleared to 0)
31:16	For LMSW, the LMSW source data For CLTS and MOV CR, cleared to 0
63:32	Reserved (cleared to 0). These bits exist only on processors that support Intel 64 architecture.

Exercise 2: Implement CR3 tracer

- Use the solution1
- Enable VM-exit for Mov-to-CR3 (process context switch) and implement a handler with logging
- Remainder:
    - Careful with subtle behavior of Mov-to-CR3 and emulate it accurately
    - See: MOV—Move to/from Control Registers

Solution

- Open exercise 2 (password: 67890)
- Enable Cr3LoadExiting in the primary processor-based VM-execution controls
- In the handler,
  - Read exit qualification,
  - Select a guest register to use as a source register
  - Make sure the bit 63 is not updated
  - Update guest’s CR3 with the value of the source register
  - Advance guest’s RIP

Wrapping Up

Security of our hypervisor

- Our hypervisor is not isolated from the guest and has no protection against ring-0
  - One can modify the hypervisor code to execute arbitrary code in VMX root operation
  - Also, can execute VMCALL and disable hypervisor
- Requires serious design and security review if you want to protect the hypervisor from ring-0

Vulnerabilities I made

- Arbitrary kernel-mode address write from ring-3
  - Background: SGDT/SIDT takes an address to write GDTR/TDTR, and executable from ring-3
  - The host must validate the address is accessible with the guest's privilege-level
- Arbitrary DR modification from ring-3
  - Background: the most of privileged instructions cause #GP **without triggering VM-exit** if privileges are insufficient
  - MOV-to-DR is an exception; it causes VM-exit and the host must check the privilege
- Unloding hypervisor from ring-3
  - Background: VMCALL is executable from ring-3

Further resources – Lightweight open source projects

- SimpleVisor by Alex Ionescu (@aionescu)
  - https://github.com/ionescu007/SimpleVisor
- Hvpp by Petr Beneš (@PetrBenes)
  - https://github.com/wbenny/hvpp
- HyperPlatform by Satoshi Tanda (@standa_t)
  - https://github.com/tandasat/HyperPlatform
- Bareflank by Rian Quinn et al.
  - https://github.com/Bareflank/hypervisor

Further resources – Heavyweight open source projects

- KVM
  - https://github.com/torvalds/linux/blob/master/arch/x86/kvm/vmx.c
- VirtualBox
  - https://www.virtualbox.org/svn/vbox/trunk/src/VBox/VMM/VMMR0/HMVMXR0.cpp
- Bhyve
  - https://github.com/freebsd/freebsd/blob/master/sys/amd64/vmm/intel/vmx.c
- Xen
  - https://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=xen/arch/x86/hvm/vmx/vmx.c;hb=HEAD
- Bochs
  - https://sourceforge.net/p/bochs/code/HEAD/tree/trunk/bochs/cpu/vmexit.cc

Further resources – Reading materials

- Hardware and Software Support for Virtualization
  - https://play.google.com/store/books/details/Hardware_and_Software_Support_for_Virtualization?id=v7Y9DgAAQBAJ
- Hypervisor From Scratch by Sinaei (@Intel80x86)
  - https://rayanfam.com/topics/hypervisor-from-scratch-part-1/

Further resources – training class

- We are holding a 5-days class for much more details!
  - Schedule: October 28 - November 1, 2019
  - Location: Fulton, Maryland (USA)
  - Trainer: Bruce Dang (@brucedang) and myself
  - https://gracefulbits.regfox.com/designing-and-implementing-a-hypervisor-for-security-analysis

Thank you!

Satoshi Tanda @standa_t / tanda.sat@gmail.com